RISC-V Computer Design

Cesar Avalos, Jacob Fustos, Stephen Longofono

December 2017

1 Introduction & Overview

This document details the design and implementation of a microcontroller containing a RISC-V core running Linux on the Nexys 4 FPGA target. The RISC-V ISA is an open-source standard for research and development of computer architectures maintained by the RISC-V foundation. In general, the ISA follows the philosophy of previous RISC implementations, favoring simplicity and regularity over specialization. RISC-V offers constructs to support all major macro and micro-architectures, and as such is a worthwhile study to complete our undergraduate education.

RISC-V is modularized into a base integer set of instructions and operations; these fundamental instructions make up the minimum viable product for any RISC-V implementation[2]. In order to implement the kernel, we will need to include a collection of extensions to the base set, dubbed "RV64G". The RV64G extension composes the minimum instruction set for use in a standard Linux system. It includes the base set of integer arithmetic instructions "I", the multiplication and division extension "M", the atomic instruction extension "A", the single-precision floating-point extension "F", and the double precision floating-point extension "D".

We intend to do the necessary work to port the Linux Kernel onto our system. This will be a great learning experience for us as this challenging process is becoming more and more common in industry. The RISC-V foundation has developed compilers, a simulator, and a number of other tools to support their Linux kernel[3]. We will need to design a first stage boot loader with device tree, reconfigure the second stage boot loader for our system, configure the kernel appropriately, and write any drivers for our devices. In order to make our core compatible with the Linux kernel, we will also need to implement an exception, trap, and interrupt system and design a modern MMU. This necessitates handling both privileged and user mode instructions.

2 Problem Constraints & Project Viability

In the interest of keeping our workload manageable, we have restricted ourselves to a serial processor with minimal caching. Designing a computer is a monumental task, and we intend to start with the most basic system before committing to additional improvements.

Specifically, our base design will include the following components:

- A processor core with integer and floating point pipelines to implement the RV64G ISA extensions.
- A control unit for the processor with control registers to implement privilege modes and traps.
- A MMU to support basic memory virtualization for both the core and the kernel.
- A serial debugger to aid kernel development and promote developer sanity.
- A VGA driver to display a terminal prompt.
- A keyboard driver to accept terminal input.
- A system bus to manage interaction of the core, the MMU, the kernel, and devices.

- A ROM unit with the bootloader code for the kernel.
- A DDR RAM controller for the MMU to control the onboard DDR banks.
- A minimal OS using BusyBox

Additional improvements discussed include caching, more sophisticated MMU, pipelined instructions, and getting Ethernet networking running. However, given the limited time we have to work on the project and our relative inexperience with computer design, these ideas are considered "stretch goals," to be pursued when the base design is complete.

With respect to financial viability, our project is inexpensive by design. Since we are implementing on an FPGA, and we intend to support common commercial, off-the-shelf peripherals, the only expenses we will incur are labor and two FPGAs for development.

We selected the Nexys4 with DDR memory from Digilent as our platform, priced at $320.00 each. This platform includes USB, RS32, and micro-USB interfaces as well as a micro-SD card reader to serve as persistent storage. The two boards are the only hardware we anticipate needing.

Our labor will vary (we will work more over the winter break), but we can estimate the number of weekly hours required to meet our goals. While school is in session, we have 5 hours in class and an additional 5 hours outside of class each week. For our three members, that puts us as 30 hours per week for the 16 week semester. Over the winter break, we will be able to devote as much as 20 hours each, for a total of 60 hours in each of the 5 weeks of the winter break. Using a local rate of $25.00 hourly for a software engineering intern, we anticipate $(30 \cdot 16 + 60 \cdot 5) \cdot \$25.00 = \$19,500.00$ in labor cost.

3 Key Milestones

Below is a description of major components to be developed in order to achieve our base design. These goals will likely evolve as we discover problems and continue to learn about the RISC-V ISA. That said, here we have outlined the major pieces of our design and their individual components. We have also included a tentative list of what parts have been delegated to what members of our team. This too may shift if any parts become too time consuming.

3.1 Major Components & Descriptions

3.1.1 Processor Core

The core processor will need to support the full RV64G extension, along with appropriate I/O to allow halting, debugging, and timing control. The general stages include a decoder, a register file, an ALU, and a write-back shift register.

3.1.2 Floating Point Unit

The floating point unit will need to work in tandem with the core, as a small pipeline running in parallel to the ALU. The control unit will need to identify FP instructions and halt the other stages of the integer pipeline until the result is ready.

3.1.3 Atomic Unit

The atomic unit will be a state machine that interacts with the control unit and MMU to perform atomic operations. The grouping of reservations and compare/swap operations is highly specialized, and should be able to be executed in full or not at all. As with the floating point unit, this specialized set of instructions will be easier to implement outside of the regular integer pipeline.

3.1.4 Control Unit

The control unit interacts with the decoder to identify the type of instructions, and distributes control signals to the stages of the core appropriately. It will include a state machine to manage privileged instructions and other control status register operations.

3.1.5 Memory Management Unit

The MMU is responsible for providing the abstraction of virtual memory, loading data from memory, and storing data to memory. It works with the system bus and kernel to move data to and from disk when necessary. It will also need direct access to the DDR4 RAM controller, so that it can load and store data for processes as needed.

3.1.6 System Bus

The system bus will handle routing of signals among the other major components in the system. It also acts as a liason to the peripheral drivers, maintaining a mapping of addresses to peripheral registers in the style of memory-mapped IO.

3.1.7 Debugger Unit

The debugger maintains developer sanity as we bring up the operating system and the kernel. The core will share the contents of its internal registers with the debugger in a read-only fashion, so that we can more effectively diagnose problems in our data and control path.

3.1.8 Peripherals

To actually use our computer as a terminal, we need a number of peripherals for human interaction. This includes software and hardware support for at minimum a VGA monitor, a keyboard, and a removable storage (USB) drive.

3.1.9 Operating System

The Operating System we will use is called BusyBox. BusyBox is a single binary meant to run on top of the Linux Kernel. Even though it is a single binary, running it in different ways has it execute the functionality of different programs such as ls, cat, and bash.

3.1.10 Kernel

The Linux kernel needs to be configured and built correctly for our design. We also need to create drivers for the individual components.

3.1.11 Bootloaders

A first stage boot loader will be needed that will be stored on a ROM chip internal to the microcontroller. This boot loader will need to know how to access the USB drive and load the Berkeley Boot Loader(BBL) into main memory. The BBL will need to be configured for our system, it handles properly loading the kernel into memory in ELF form as opposed to cutting it up into a flat binary.

3.2 Component Design & Tradeoffs

This section serves as a work-in-progress overview of our design and plans for individual components. These individual components serve the needs of the major components defined above. Note that only the parts we have designed thus far are described; the remaining components will be designed once we have a better idea of the software and hardware constraints of these major pieces.

In many ways, our biggest tradeoff concern is developer time and experience. This project is as much about implementing a RISC-V computer as it is about learning how the disparate parts we have studied come together. As such, we will need to balance our ambitions with what we have the ability to learn and implement in a year's time. To address this, we have simplified and modularized our designs such that they are relatively easy to implement and test. Needless to say, this comes at the expense of overall performance. The plan is to develop a working, simple design and iterate on it. In this way, we can ensure a working design at the end, and implement optimizations after the fact without compromising our entire project.

3.2.1 The Core

The main core will follow the lead of the base integer core as presented in the RISC-V textbook [1]. Figure 1 depicts the high-level flow of instructions and data through the core. The control unit maintains control lines for all other modules, to dictate what action should be taken at any given time (including halting). The control unit and the MMU work together to queue, execute, and write back instructions and their results.

When an instruction is ready, the decoder identifies its type and breaks out the addresses, function and opcodes, and immediate values. The register file coordinates reading and writing to the standard set of 64-bit general purpose registers. The ALU, FPU, and atomic modules handle operations on the data loaded through the register file, and feed the results into a write back shift register which halts until the MMU is ready to write back the results.

As noted above, the complex nature of floating point operations is such that the floating point module must be blocking. Any given floating point operation might take several cycles to complete, and the standard floating point instruction set includes multiple-operation instructions. If an instruction comes in after a floating point result it will be halted until a valid result is ready at the FPU.

Likewise, the atomic unit instructions will span several clock cycles, and must by definition complete in full before any other instruction is executed. Rather than try to coerce the integer pipeline into orchestrating such an event, we opted to perform atomic instruction groups via a state machine.

At the control level, the control unit and the control status registers will be used to track the privilege mode and handle privileged instructions. This is closely related to the handling of interrupts and traps, so the control status registers and the control unti will be transparent to an interrupt handler module. This will be a state machine that is capable of flushing the pipeline and interacting with the MMU to handle interrupts as outlined in the RISC-V specifications.

Finally, to facilitate development and troubleshooting, the core will have asynchronous buses with the contents of the register and control status registers fed to the debugger unit. If we run into issues, this gives us a play-by-play recap of what went through the core after we have moved it out of simulation.

3.2.2 The Debugger

The debugger has three main components. The debugger uses UART to interface with the PC, as such the first and second components are the UART RX and TX components (receive and transmit respectively). The third component is the debug controller, which takes care of the debugging logic. The debugger unit receives from the core the registers and control status, and the debugger sends a pause signal to the core. The idea behind the core is very simple, the user sends a command, the RX component receives said command, the debugger logic identifies said command and prepares the action to be performed, the action is performed, and finally the TX responds back with the outcome. It should be noted that some complications might arise if the processor is pipelined along the way, and the design must be modified accordingly.

3.2.3 Spike

The RISC-V Simulator(Spike) has been gone through and understood. This was for 2 reasons: one it is a functional core, so we have been able to learn much about RISC-V implementations that support the full 64G extensions, it also will work for development while the board is in the early stages.

3.2.4 The Berkeley Bootloader

The current software task is to look through the BBL to confirm that it is working properly on the simulator. While it is being gone through we shall take notes as what we need to do to modify it for our system. Currently it is built.

4 Tentative Task List & Summary of Work to Date

We have roughly divided the work among ourselves as follows. Cesar will work on the system bus, peripheral drivers, and the debugger. Jacob will work on the Linux kernel, the OS and its bootloader, and the

software size of the debugger. Stephen will work on the core implementation, and integration over privileged instruction with the OS.

Table 1: Task List

Item	Delegated To	Status	Date Completed
Decoder	Stephen	Complete & tested	11/15/17
Register File	Stephen	Complete & tested	11/27/17
Integer ALU	Stephen	WIP	12/20/17
Atomic Unit	Stephen	-	
FP ALU	Stephen	Researching	
Control Unit	Stephen	-	
Debugger	Cesar	WIP	
System Bus	Cesar	Researching	
Keyboard Controller	Cesar	-	
Keyboard Driver	Jacob	-	
VGA Controller	Cesar	-	
VGA Driver	Jacob	-	
USB Controller	Cesar	-	
USB Driver	Jacob	-	
DDR Controller	Cesar	Researching	
DDR Driver	Jacob	-	
Minimal Kernel	Jacob	WIP	
System Bootloader	Jacob	-	
Berkeley Bootloader	Jacob	Researching	
OS Built	Jacob	WIP	
MMU	Team	-	

5 Design Documents

See Figure 1, a depiction of the core design to date.

6 Conclusion & Executive Summary

We have chosen a very ambitious project. As mentioned above, we must be diligent with respect to our goals and deadlines. While this project could be very beneficial for achieving our future academic and industry goals, it is also important to learn how to plan and execute a large project. We are familiar with the concepts of many of the components that we are working with, but actually getting into the weeds of implementing them and integrating them together will take us to our limits. We acknowledge that, in the end, what will bring this project together is the guidance of a wise beyond his years sage whose enthusiasm for the project will keep us motivated.

References

[1] David Patterson and John Hennessy. *Computer Organization and Design RISC-V Edition: The Hardware Software Interface*. Morgan Kaufmann, Burlington, MA, 2017.

[2] David Patterson and Andrew Waterman. *The RISC-V Reader: An Open Architecture Atlas*. Strawberry Canyon, Berkeley, CA, 2017.

[3] RISC-V Foundation. *Software Status*. URL: `https://riscv.org/software-status/`.

Figure 1: Data Flow Diagram, Processor Core